COMP 550
Algorithm and Analysis

Correctness and Running Time

Based on CLRS Sec 2.1, 2.2

Some slides are adapted from ones by prior instructors Prof. Plaisted and Prof. Osborne

Algorithm: More Formal

- A <u>finite</u> sequence of <u>rigorous</u> instructions for solving a well-specified computational problem

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements

Output: Maximum element of A

```
CurrentMax = A[1]
for i = 2 to n do
    if A[i] > CurrentMax then
        CurrentMax = A[i]
return CurrentMax
```

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements

Output: Maximum element of A

```
Assign the maximum element of A in CurrentMax
return CurrentMax
```

Algorithm: Formal

- Alonzo Church and Alan Turing in 1936 came with formal definitions for the concept of algorithm
- One definition: Turing Machine that *always* halts.
- Other definitions: Lambda Calculus, Recursive Functions
- These definitions are equivalent among each others

Algorithm Analysis

- If we can develop an algorithm for a problem, we need to analyze it (note that some problem may be unsolvable!)
- Is the algorithm correct?
  - Use mathematical tools
- How efficient it is?
  - Why needed? How to measure this?

Algorithm Correctness Proof

FindMax: Correctness

Algorithm 1 FindMax(A, n)

Input: Array A of n elements
Output: Maximum element of A

```
CurrentMax = A[1]
for i = 2 to n do
    if A[i] > CurrentMax then
        CurrentMax = A[i]
return CurrentMax
```

- FindMax is correct iff
  - For **each possible input**, FindMax **halts** and returns **the maximum of input array**

FindMax: Correctness

- Intuitive explanation $\neq$ Correctness proof
- An example where the algorithm works $\neq$ Correctness proof
- Common proof techniques
  - Proof by Construction
  - Proof by Induction
  - Proof by Contradiction

FindMax: Correctness

- **Loop invariant**
  - A property that is true before, during, and after a loop.
- Proving a loop invariant: 3 steps
  - **Initialization**: the loop invariant is true before the loop starts.
  - **Maintenance**: if the invariant is true before one loop iteration, it remains true before the next.
  - **Termination**: The loop terminates AND the invariant gives a useful property that helps show why the algorithm is correct.

FindMax: Correctness

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements
Output: Maximum element of A

```
1: CurrentMax = A[1]
2: for i = 2 to n do
3:     if A[i] > CurrentMax then
4:         CurrentMax = A[i]
5: return CurrentMax
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 2-4, CurrentMax contains the maximum of all elements in the subarray A[1 : i-1].

FindMax: Correctness

Algorithm 1 FindMax(A, n)

Input: Array A of n elements
Output: Maximum element of A

```
1: CurrentMax = A[1]
2: for i = 2 to n do
3:     if A[i] > CurrentMax then
4:         CurrentMax = A[i]
5: return CurrentMax
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 2-4, CurrentMax contains the maximum of all elements in the subarray A[1 : i-1].

Initialization

LI holds before the first iteration (when i=2)

LI with i=2: CurrentMax contains the maximum of the subarray A[1 : 1].

Proof: Due to line 1.

FindMax: Correctness

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements
Output: Maximum element of A

```
1: CurrentMax = A[1]
2: for i = 2 to n do
3:     if A[i] > CurrentMax then
4:         CurrentMax = A[i]
5: return CurrentMax
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 2-4, CurrentMax contains the maximum of all elements in the subarray A[1 : i-1].

Maintenance

LI holds before the (k+1)-th iteration assuming that it holds before the k-th iteration.

If CurrentMax contains the maximum of the subarray A[1 : k-1] before the k-th iteration, then CurrentMax contains the maximum of the sub-array A[1 : k] before the (k+1)-th iteration.

FindMax: Correctness

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements
Output: Maximum element of A

```
1: CurrentMax = A[1]
2: for i = 2 to n do
3:     if A[i] > CurrentMax then
4:         CurrentMax = A[i]
5: return CurrentMax
```

Maintenance: If CurrentMax contains the maximum of the subarray A[1 : k-1] before the k-th iteration, then CurrentMax contains the maximum of the sub-array A[1 : k] before the (k+1)-th iteration.

Proof: We need to consider the k-th iteration.

- CurrentMax holds $\max_{1 \le i \le k-1} A[i]$ before the k-th iteration (induction hypothesis).
- Case 1: $\forall i \le k-1, A[i] < A[k]$ holds.
  - Then, $\max_{1 \le i \le k-1} A[i] < A[k]$
  - By induction hypothesis, CurrentMax $< A[k]$
  - Line 4 assigns $A[k]$ to CurrentMax
  - CurrentMax $= \max_{1 \le i \le k} A[i]$

FindMax: Correctness

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements
Output: Maximum element of A

```
1: CurrentMax = A[1]
2: for i = 2 to n do
3:     if A[i] > CurrentMax then
4:         CurrentMax = A[i]
5: return CurrentMax
```

Maintenance: If CurrentMax contains the maximum of the subarray A[1 : k-1] before the k-th iteration, then CurrentMax contains the maximum of the sub-array A[1 : k] before the (k+1)-th iteration.

Proof: We need to consider the k-th iteration.

- CurrentMax holds $\max_{1 \le i \le k-1} A[i]$ before the k-th iteration (induction hypothesis).
- Case 2: $\exists i \le k-1, A[i] \ge A[k]$ holds.
  - Then, $\max_{1 \le i \le k-1} A[i] \ge A[k]$
  - By induction hypothesis, $\text{CurrentMax} \ge A[k]$
  - Line 3 condition is False
  - $\text{CurrentMax} = \max_{1 \le i \le k} A[i]$
- i incremented by 1

FindMax: Correctness

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements
Output: Maximum element of A

1: CurrentMax $= A[1]$
2: **for** $i = 2$ to n **do**
3:     **if** $A[i] >$ CurrentMax **then**
4:         CurrentMax $= A[i]$
5: **return** CurrentMax

Loop invariant

(LI) At the start of each iteration of the for loop of lines 2-4, CurrentMax contains the maximum of all elements in the subarray $A[1 : i\text{-}1]$.

Termination

Loop terminates when $i > n$.

i is an integer → loop terminates when $i = n+1$

At the start of (n+1)-th iteration of the for loop of lines 2-4, CurrentMax contains the maximum of all elements in the subarray $A[1 : n]$.

FindMax: Correctness

Algorithm 1 FINDMAX(A, n)

Input: Array A of n elements
Output: Maximum element of A

1: CurrentMax $= A[1]$
2: **for** $i = 2$ to n **do**
3:     **if** $A[i] >$ CurrentMax **then**
4:         CurrentMax $= A[i]$
5: **return** CurrentMax

Loop invariant

(LI) At the start of each iteration of the for loop of lines 2-4, CurrentMax contains the maximum of all elements in the subarray $A[1 : i\text{-}1]$.

We also need to prove that the algorithm always halts.

- Trivial here
  - i cannot increase beyond $n + 1$
- Not always easy!

Algorithm Efficiency

- There are often many approaches (algorithms) to solve a problem (Recall: Finding the maximum)
  - How do we choose between them?
- At the heart of computer program design are two (sometimes conflicting) goals
  1. To design an algorithm that is easy to understand, code, debug
  2. To design an algorithm that makes efficient use of the computer's resources

Algorithm Efficiency

Algorithm Efficiency

- (1) is the concern of Software Engineering
- (2) is the concern of Algorithm Analysis
- Following questions are relevant for (2):
  - How to find the most efficient of several possible algorithms for the same problem.
  - Is the algorithm optimal (best in some sense)?
    - Can we do even better?

Algorithm Efficiency

- Since (2) is about efficiency, what should be the metric to determine efficiency?
  - Computation time (a.k.a. running time)
  - Memory requirement
  - Communication bandwidth, etc.
- Primary concern:

  i) computation time, ii) memory requirement

Determining Running Time

- Option 1: Empirical analysis (run executable code)
- Option 2: Theoretical analysis

- Which one is better?

Option 1: Empirical Analysis

- Run executable code
- Compare **Alg. A** and **Alg. B** that solve the same problem
  - Need to implement both
  - Suppose Alg. A shows takes less computation time (avg or worst?)
    - Alg. A might be better coded
    - Can run both on finite number of test cases (inputs). What if Alg. A is suitable for these inputs, but does poorly in many unseen inputs?
    - The computing platform may favor Alg. A

Option 2: Theoretical Analysis

- What computing model needs to be assumed?
- What should be the costs of different operations?
- Running time should be determined with respect to what?
- An algorithm can be represented differently, how to make the analysis independent of this dissimilarity?

Computing Model

- Running time measurement should be machine independent
  - Use a hypothetical computing platform (should not be overly unrealistic)
- Our assumption: Random Access Machine (RAM) model
  - Single processor
  - Executes instructions one after another (no concurrency)

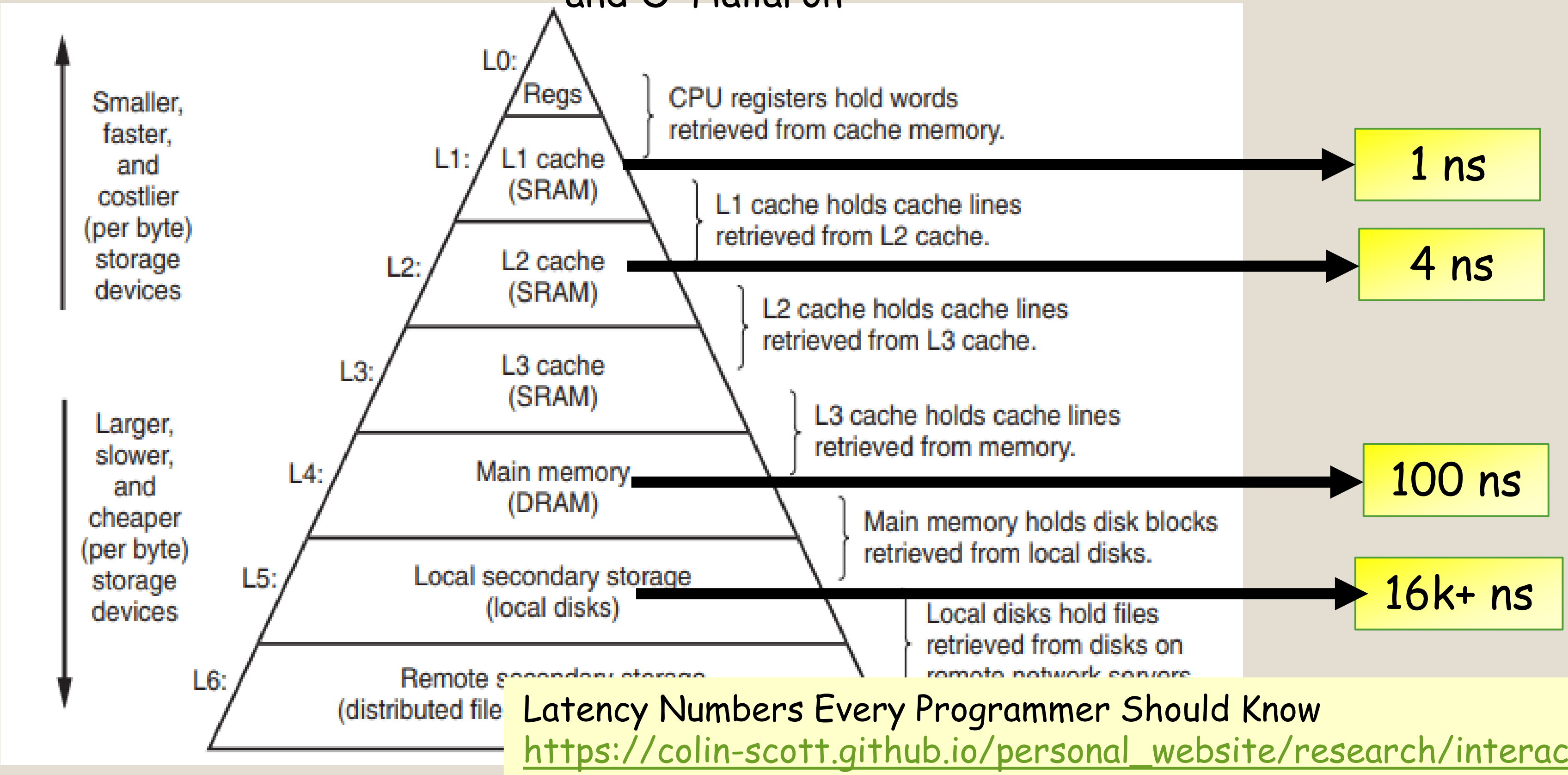
RAM Model

- Supports primitive constant-time instructions
  - Arithmetic (+, -, *, /, %, floor, ceiling).
  - Data Movement (load, store, copy).
  - Control (branch, subroutine call).
- No complex operations supported
  - No sort operation (can't assume to do it in constant time)
- Simplifying assumption: run time is 1 time unit for all simple instructions.

RAM Model

- Memory is unlimited
- No memory hierarchy

Fig. Source: Computer Systems: A Programmer's Perspective, Bryant and O' Hallaron



Latency Numbers Every Programmer Should Know
https://colin-scott.github.io/personal_website/research/interactive_latency.html

Running Time

- The **running time** of an algorithm on a particular input is the **number of primitive operations** or "steps" executed
  - Also called "**time complexity**" of the algorithm
- Running time usually grows with "**input size**"
  - Steps required to sort 1 million numbers vs 1000 numbers
- Running time also depends on other input characteristics
  - Sorting an already sorted array

Input Size

- Determine running time w.r.t. input size
- Formally, input size depends on how input is encoded
  - We'll see this later in this course
- Input size depends on the problem in hand
  - Array problems: number of items
  - Graph problems: number of vertices and edges

Worst, Average, and Best-Case Complexity

- Worst-case complexity
  - **Maximum** steps the algorithm takes for any possible input
  - Most tractable measure
- Average-case complexity
  - ***Average*** number of steps for all possible inputs
  - Requires probability distribution of possible inputs, which is usually difficult to provide and to analyze
- Best-case complexity
  - **Minimum** number of steps for any possible input
  - Not useful. **Why?**

Why Worst-Case Complexity?

- An upper bound on the running time for any input
  - The algorithm never takes any longer
- For some algorithms, the worst case occurs fairly often
- The average case is often roughly as bad as the worst case

Worst-Case Complexity: Example

FindMax(A,n)	Cost	Times
`CurrentMax ← A[1]`		
`for i ← 2 to n do`		
`    if A[i] > CurrentMax then`		
`        CurrentMax ← A[i]`		
`return CurrentMax`		

Worst-Case Complexity: Example

FindMax(A,n)	Cost	Times
`CurrentMax ← A[1]`	c_1	1
`for i ← 2 to n do`	c_2	
`   if A[i] > CurrentMax then`	c_3	
`      CurrentMax ← A[i]`	c_4	
`return CurrentMax`	c_5	

Worst-Case Complexity: Example

	FindMax(A,n)	Cost	Times
1.	`CurrentMax ← A[1]`	c_1	1
2.	`for i ← 2 to n do`	c_2	n
3.	`    if A[i] > CurrentMax then`	c_3	
4.	`        CurrentMax ← A[i]`	c_4	
5.	`return CurrentMax`	c_5	

Worst-Case Complexity: Example

	FindMax(A,n)	Cost	Times
1.	`CurrentMax ← A[1]`	c_1	1
2.	`for i ← 2 to n do`	c_2	n
3.	`    if A[i] > CurrentMax then`	c_3	$n-1$
4.	`        CurrentMax ← A[i]`	c_4	
5.	`return CurrentMax`	c_5	

In the **worst-case**, how many times line 4 executes?

Worst-Case Complexity: Example

FindMax(A,n)	Cost	Times
`CurrentMax ← A[1]`	c_1	1
`for i ← 2 to n do`	c_2	n
`    if A[i] > CurrentMax then`	c_3	$n-1$
`        CurrentMax ← A[i]`	c_4	$n-1$
`return CurrentMax`	c_5	1

Worst-case running time, $T(n)$

$$
\begin{aligned}
&= c_1 + c_2 \cdot n + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \\
&= (c_2 + c_3 + c_4) \cdot n + c_1 - c_3 - c_4 + c_5 \\
&= a \cdot n + b
\end{aligned}
$$

Best-Case Complexity: Example

	FindMax(A,n)	Cost	Times
1.	`CurrentMax ← A[1]`	c_1	1
2.	`for i ← 2 to n do`	c_2	n
3.	`    if A[i] > CurrentMax then`	c_3	$n-1$
4.	`        CurrentMax ← A[i]`	c_4	?
5.	`return CurrentMax`	c_5	1

In the **best-case**, how many times line 4 executes?

Best-Case Complexity: Example

FindMax(A,n)	Cost	Times
`CurrentMax ← A[1]`	c_1	1
`for i ← 2 to n do`	c_2	n
`    if A[i] > CurrentMax then`	c_3	$n-1$
`        CurrentMax ← A[i]`	c_4	0
`return CurrentMax`	c_5	1

Best-case running time, $T(n)$

$$\begin{aligned} T(n) &= c_1 + c_2 \cdot n + c_3 \cdot (n-1) + c_4 \cdot 0 + c_5 \\ &= (c_2 + c_3 + c_4) \cdot n + c_1 - c_3 + c_5 \\ &= a \cdot n + b \end{aligned}$$

Average-Case Complexity: Example

FindMax(A,n)	Cost	Times
1. `CurrentMax ← A[1]`	c_1	1
2. `for i ← 2 to n do`	c_2	n
3. `    if A[i] > CurrentMax then`	c_3	$n-1$
4. `        CurrentMax ← A[i]`	c_4	x
5. `return CurrentMax`	c_5	1

Avg-case running time, $T(n) = c_1 + c_2 \cdot n + c_3 \cdot (n-1) + c_4 \cdot E[x] + c_5$

$$= (c_2 + c_3) \cdot n + c_1 - c_3 + c_4 \cdot \frac{\sum_{i=0}^{n-1} i}{n} + c_5$$

Insertion Sort: Time Complexity

Goal: evaluate $\sum_{i=0}^{n-1} i$

$$\sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

The summation

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

is an *arithmetic series* and has the value

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Evaluating the sums

- Appendix A: Summations

Average-Case Complexity: Example

	FindMax(A,n)	Cost	Times
1.	`CurrentMax ← A[1]`	c_1	1
2.	`for i ← 2 to n do`	c_2	n
3.	`    if A[i] > CurrentMax then`	c_3	$n-1$
4.	`        CurrentMax ← A[i]`	c_4	x
5.	`return CurrentMax`	c_5	1

Avg-case running time, $T(n) = c_1 + c_2 \cdot n + c_3 \cdot (n-1) + c_4 \cdot E[x] + c_5$

$$= (c_2 + c_3) \cdot n + c_1 - c_3 + c_4 \cdot \frac{\sum_{i=0}^{n-1} i}{n} + c_5$$

$$= (c_2 + c_3) \cdot n + c_1 - c_3 + c_4 \cdot \frac{(n-1) \cdot n}{n \cdot 2} + c_5$$

$$= a \cdot n + b$$

Searching Problem

- Find an element in a sequence of numbers
  - **Input**: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ and a key
  - **Output**: True if $\exists k \in \{1..n\} : key = a_k$, False otherwise
  - *Example:*

    Input: $\langle 31, 41, 59, 26, 41, 58 \rangle, 59$

    Output: True

Linear Search

LinearSearch(A,n,key)	Cost	Times
i = 1		
while i ≤ n and A[i] ≠ key do		
i = i + 1		
if i ≤ n then		
return True		
return False		

Try at home: Correctness proof by loop invariant.

Linear Search: Worst-Case

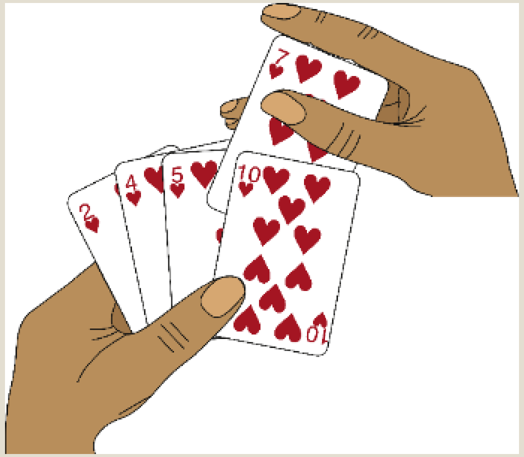
LinearSearch(A,n,key)	Cost	Times
`i = 1`	c_1	1
`while i ≤ n and A[i] ≠ key do`	c_2	x
`    i = i + 1`	c_3	$x-1$
`if i ≤ n then`	c_4	1
`    return True`	c_5	1
`return False`	c_6	1

x is an integer between 1 and $n+1$

- Worst case: $x = n + 1$
- Best case: $x = 1$
- Average case: $x = n/2$

Insertion Sort

- Informal description:
  - Iterate the array from the first element
  - If the current element is in wrong place w.r.t. already seen elements, move it to its correct place



Insertion Sort

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i – 1].
    j = i – 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key
```

From CLRS 4th edition

Insertion Sort

```
INSERTION-SORT(A, n)
  for i = 2 to n
      key = A[i]
      // Insert A[i] into the sorted subarray A[1 : i − 1].
      j = i − 1
      while j > 0 and A[j] > key
          A[j + 1] = A[j]
          j = j − 1
      A[j + 1] = key
```

Insertion Sort: Correctness

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i - 1].
4      j = i - 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j - 1
8      A[j + 1] = key
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 1-8, the subarray A[1 : i-1] consists of all elements originally in A[1 : i-1] and **A[1 : i-1] is sorted**.

Insertion Sort: Correctness

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 1-8, the subarray $A[1:i\text{-}1]$ consists of all elements originally in $A[1:i\text{-}1]$ and $\mathbf{A[1:i\text{-}1]}$ **is sorted.**

Initialization

LI holds before the first iteration (when i=2)

LI with i=2: The subarray $A[1:1]$ consists of all elements originally in $A[1:1]$ and $\mathbf{A[1:1]}$ **is sorted.**

Proof: Trivially true.

Insertion Sort: Correctness

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 1-8, the subarray A[1 : i-1] consists of all elements originally in A[1 : i-1] and **A[1 : i-1] is sorted.**

Maintenance

LI holds before the (k+1)-th iteration assuming it holds before the k-th iteration.

If the subarray A[1 : k-1] is sorted before the k-th iteration, then A[1 : k] is sorted before the (k+1)-th iteration.

Insertion Sort: Correctness

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

Maintenance: If the subarray $A[1:k]$ is sorted before the k-th iteration, then $A[1:k+1]$ is sorted before the (k+1)-th iteration.

Proof: We need to consider the k-th iteration.

- A[1:k-1] is already sorted before that.
- Let $k^* < k$ be the last index such that $A[j] \le A[k]$ for all $j \le k^*$. (Loop breaks at $j=k^*$).
  - $k^* = 0$ if $A[j] > A[k]$ for all $j < k+1$.
- Lines 5-7 shifts each A[j] with $k^* < j < k$ to one position right.
- Line 8 moves key to $A[k^*+1]$.

Informal!

- i increments by 1

Insertion Sort: Correctness

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

Loop invariant

(LI) At the start of each iteration of the for loop of lines 1-8, the subarray $A[1 : i\text{-}1]$ consists of all elements originally in $A[1 : i\text{-}1]$ and **$A[1 : i\text{-}1]$ is sorted**.

Termination

Loop terminates when $i > n$.

i is an integer → loop terminates when $i=n+1$

At the start of $(n+1)$-th iteration of the for loop of lines 1-8, the subarray $A[1 : n]$ consists of all elements originally in $A[1 : n]$ and **$A[1 : n]$ is sorted**.

Insertion Sort: Time Complexity

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	
$key = A[i]$	c_2	
// Insert $A[i]$ into the sorted subarray $A[1 : i-1]$.	0	
$j = i - 1$	c_4	
while $j > 0$ and $A[j] > key$	c_5	
$A[j+1] = A[j]$	c_6	
$j = j - 1$	c_7	
$A[j+1] = key$	c_8	

t_i:
How many **while** loop tests for current value of i

Running time, $\mathrm{T}(n) = c_1 \cdot n + c_2(n-1) + c_4(n-1) + c_5 \cdot \sum_{i=2}^{n} t_i + c_6 \cdot \sum_{i=2}^{n}(t_i - 1) + c_7 \cdot \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$

$$= (c_1 + c_2 + c_4 + c_8)\, n + c_5 \cdot \sum_{i=2}^{n} t_i + (c_6 + c_7) \sum_{i=2}^{n}(t_i - 1) - (c_2 + c_4 + c_8)$$

$$= a \cdot n + c_5 \cdot \sum_{i=2}^{n} t_i + (c_6 + c_7) \sum_{i=2}^{n}(t_i - 1) + b$$

Insertion Sort: Time Complexity

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i - 1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n-1$

t_i:
How many **while** loop tests for current value of i

Question: When do the best and worst cases occur?

Insertion Sort: Time Complexity

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $\quad key = A[i]$	c_2	$n-1$
3 $\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $\quad j = i - 1$	c_4	$n-1$
5 $\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $\quad\quad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $\quad A[j+1] = key$	c_8	$n-1$

t_i:
How many **while** loop tests for current value of i

Best Case: Already sorted array. Why?

Lines 6,7 never executes. $\boldsymbol{t_i = 1}$

In the best case, $\mathrm{T}(n) = a \cdot n + c_5 \cdot \sum_{i=2}^{n} t_i + (c_6 + c_7) \sum_{i=2}^{n} (t_i - 1) + b$

$$= a \cdot n + c_5 \cdot (n-1) + b = a' \cdot n + b'$$

Insertion Sort: Time Complexity

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $\quad key = A[i]$	c_2	$n-1$
3 $\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $\quad j = i - 1$	c_4	$n-1$
5 $\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $\quad\quad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $\quad A[j+1] = key$	c_8	$n-1$

t_i:
How many **while** loop tests for current value of i

Worst Case: Reverse sorted array. Why?

Lines 5 executes until $j = 0$. So, executes for $\boldsymbol{j = i - 1, i - 2, \ldots, 0}$.

$$\boxed{t_i = i}$$

So, line 5 executes $\sum_{i=2}^{n} i$, lines 6 and 7 each execute $\sum_{i=2}^{n} (i-1)$ times.

Insertion Sort: Time Complexity

Goal: evaluate $\sum_{i=2}^{n} i$ and $\sum_{i=2}^{n}(i-1)$

$$\sum_{i=2}^{n} i = \sum_{i=1}^{n} i - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

The summation

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

is an *arithmetic series* and has the value

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Evaluating the sums

- Appendix A: Summations

Insertion Sort: Time Complexity

In the worst case,

$$\mathrm{T}(n) = a \cdot n + c_5 \cdot \sum_{i=2}^{n} t_i + (c_6 + c_7) \sum_{i=2}^{n} (t_i - 1) + b$$

$$= a \cdot n + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \left(\frac{n(n-1)}{2} \right) + b$$

$$= (\frac{c_5 + c_6 + c_7}{2}) n^2 + \left(\mathrm{a} + \frac{c_5 - c_6 - c_7}{2} \right) n + b - c_5$$

$$= a' n^2 + b' n + c'$$

Order of Growth

- Principal interest is to determine
  - How running time grows with input size: **Order of growth**.
  - The running time for large inputs: **Asymptotic complexity**.
- Running time growth as input size goes to ∞
  - Lower-order terms and coefficient of the highest-order term are insignificant.
  - In $3n^3+7n+1$, which term dominates the running time for very large n?
    - Above running time is $\Theta(n^3)$

Order of Growth

- Express the worst- and best-case running times of
  - FindMax
  - Linear Search
  - Insertion Sort

Thank You!